Improving Processor Availability in the MPI Implementation for the ASCI/Red Supercomputer

Ron Brightwell, William Lawry
Scalable Computing Systems Department
Sandia National Laboratories†
P.O. Box 5800
Albuquerque, NM, 87111-1110
bright,wflawry @cs.sandia.gov

Arthur. B. Maccabe, Christopher Wilson
Scalable Systems Lab
Computer Science Department
University of New Mexico
Albuquerque, NM, 87131-1386
maccabe,riley @cs.unm.edu.

Abstract

This paper describes how a portable benchmark suite that measures the ability of an MPI implementation to overlap computation and communication can be used to discover and diagnose performance problems. We describe the approach of the benchmark suite and discuss a performance problem that we uncovered with the MPI implementation on the ASCI/Red supercomputer. A slight modification to the MPI implementation has resulted in a significant gain CPU availability and bandwidth with a slight degradation in latency performance. We present a detailed analysis of these results and discuss how the benchmark suite has enabled us to tailor the MPI implementation to optimize for all three measurements.

Keywords: System-area network, Message-passing, MPI, Performance Analysis

1 Introduction

We have designed and implemented a portable benchmark suite called COMB, the Communication Offload MPI-based Benchmark, that measures the ability of an MPI implementation to overlap computation and MPI communication. The ability to overlap computation with communication is influenced by several system characteristics, such as the quality of the MPI implementation and the capabilities of the underlying network transport layer. For example, some message passing systems interrupt the host CPU to obtain resources from the operating system in order to receive packets from the network. This strategy is likely to adversely impact the utilization of the host CPU, but may allow for an increase in MPI bandwidth.

While our benchmark was developed to measure the quality of MPI implementations on clusters, during the initial development of the benchmark suite, we made several runs on the ASCI/Red supercomputer at Sandia National Laboratories. Initially the runs were made to validate the results of the benchmark suite on a tightly-coupled parallel platform. However, the benchmark suite revealed a subtle but significant performance problem with the MPI implementation. In relating these results and a subsequent change to the implementation to correct the problem, we demonstrate that the benchmark suite can provide greater insight into the relationship between network performance and CPU performance.

The rest of this paper is organized as follows. In the next section, we provide general background for our interest in processor availability. Section 3 describes the benchmark suite. In Section 4, we provide an overview of the hardware and software environment of the ASCI/Red supercomputer. Section 5 presents initial results from the benchmark suite and describes the performance problem that was revealed. This section continues by describing a small MPI enhancement and the resulting performance impact. Section 6 discusses the conclusions of this paper.

This work was supported in part through the Computer Science Research Institute (CSRI) at Sandia National Laboratories under contract number SF-6432-CR.

† Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

```
Producer:                          Consumer:
double A[BSIZE], B[BSIZE];         double A[BSIZE], B[BSIZE];

fill A;                            ireceive A; isend CTS A;
wait CTS A; isend A;

fill B;                            ireceive B; isend CTS B;
wait CTS B; isend B;

for( i = 0 ; i ¡ n-1 ; i++ )       for( i = 0 ; i ¡ n ; i++ )
   wait A sent; fill A;               wait A received; sum A;
   wait CTS A; isend A;               ireceive A; isend CTS A;

   wait B sent; fill B;               wait B received; sum B;
   wait CTS B; isend B;               ireceive B; isend CTS B;
```

Figure 1. Pseudocode for Double Buffering

2 Background

Like most supercomputers, ASCI/Red and the systems software for ASCI/Red was developed to support "resource constrained applications," applications for which the problem size can be scaled to consume all of one or more of the resources provided by the computing system. That is, the size of the problem is constrained by the availability of specific resources. In many cases, these applications are constrained by the availability of processor cycles. The ability to manage processor cycles is critical for these applications.

To support application programmers in their efforts to manage processor cycles, the MPI standard includes non-blocking send and receive operations. These operations are included in the standard to permit overlap between computation and communication. In particular, an application programmer can initiate a non-blocking communication operation (either a send or a receive) and continue with a meaningful part of their computation while the communication progresses. Later, the application can poll for the completion of the communication. If the standard only provided blocking operations, the processor cycles during communication would not be available to the application and would be wasted.

2.1 Double Buffering

Perhaps the simplest example of overlapping computation and communication involves the use of double buffering. In an earlier experiment[6], we measured the time taken to produce and sum a long stream of floating point numbers. In this experiment, the application consists of two processes, a *producer* and a *consumer*. The producer uses a random number generator to produce a stream of double precision floating point values and the consumer calculates the sum of the values it receives. The producer prepares a batch of numbers which are then sent to the consumer. The consumer provides two buffers so that the producer can fill one buffer while the consumer is processing the other buffer. Pseudo-code for the producer and consumer processes is presented in Figure 1.

In examining this code, notice that the producer uses non-blocking sends to transmit filled buffers. This should allow the producer to overlap its filling of one buffer with the sending of the previously filled buffer. Similarly, the consumer uses pre-posted, non-blocking receives. When the producer is faster than the consumer, this should allow the consumer to overlap the processing of one buffer with the reception of the next buffer.

Our experiments compared MPICH/GM[7] with an implementation of MPI over Portals[2]. Even though the MPICH/GM communication bandwidth was substantially higher (80 MB/s versus 50 MB/s), the overall processing rate was significantly better, 15%, when we used MPI over Portals. The improvement is due to the fact that the MPI over Portals implementation allows complete overlap of computation with communication.

2.2 Availability and Utilization

The reader may note that we present our discussion and results in terms of processor *availability*, how much of the processor is available to the application, rather than processor *utilization*, how much of the processor is utilized during communication. The two terms are effectively inverses of one another, that is, high availability implies low utilization and vice versa. While it is common practice to report processor utilization, we find that utilization sends the wrong message. Utilization seems to be a good thing and one naturally assumes that higher utilization is better when, in fact, the opposite is actually the case.

Ultimately, the difference is one of perspective. We find that using the term availability keeps us focused on the fact that we are trying to provide resources to applications.

3 The COMB Benchmark Suite

In 1999, White and Bova[10] noted that many MPI implementations do not support overlapping computation with communication, even though it is clear that the MPI standard intended that implementations support this overlap. Given our experience in the double buffering experiment and the observation of White and Bova, we set out to measure the degree to which MPI implementations supported overlap between computation and communication.

The COMB benchmark[5] suite consists of two different methods for measuring the performance of a system, each with a different perspective on characterizing the ability to overlap computation and MPI communication. This multi-method approach captures performance data on a wider range of the system and allows for results from each benchmark to be validated and/or reinforced by the other. The first method, the *Polling Method*, allows for the maximum possible overlap of computation and MPI communication. The second method, the *Post-Work-Wait Method* tests for overlap under practical restrictions on MPI calls.

3.1 Polling Method

The *polling method* uses two processes, one process, the *worker process*, counts cycles and performs message passing. A second, *support process*, runs on the second node and only performs message passing. Figure 2 presents pseudo code for the worker process. All receives are posted before sends. Initial setup of message passing as well as conclusion of same are omitted from the figure. Additionally, Figure 3 provides a pictorial representation of the method.

```
read current time
for( i = 0 ; i < work/poll_factor ; i++ )
   for( j = 0 ; j < poll_factor ; j++)
      /* nothing */

   if(asynchronous receive is complete)
      start asynchronous reply(s)
      post asynchronous receive(s)


read current time
```

Figure 2. Polling Method Psuedocode For Worker Process

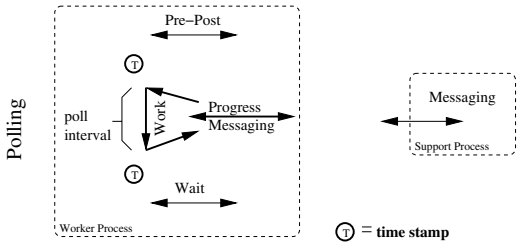

Figure 3. Overview of Polling Method

This method uses a ping-pong communication strategy with messages flowing in both directions between sender node and receiver. Each process polls for message arrivals and propagates replacement messages upon completion of earlier messages. After a predetermined amount of computation, bandwidth and CPU availability are computed. The polling interval can be adjusted to demonstrate the trade-off between bandwidth and CPU availability. Because this method never blocks waiting for message completion it provides an accurate report of CPU availability.

As can be seen in Figure 2, after a fixed number of iterations in the inner loop the worker process polls for receipt of the next message. The number of iterations of the inner loop determines the time between polls and, hence, determines the polling interval. If a test for completion is negative, the worker process will iterate through another polling interval before testing again. If a test for completion is positive, the process will post related messaging calls and will similarly address any other received messages before entering another polling

interval. The support process sends messages as fast as they are consumed by the receiver.

We vary the polling interval to elicit changes in CPU availability and bandwidth. When the polling interval becomes sufficiently large all possible message transfers may complete during the polling interval and communication then must wait, resulting in decreased bandwidth.

The polling method uses a queue of messages at each node in order to maximize achievable bandwidth. When either process detects that a message has arrived, it iterates through the queue of all messages that have arrived, sending replies to each of these messages. When we set the queue size to one, a single message passed between the two nodes then the polling method acts as a standard ping-pong test and maximum sustained bandwidth will be sacrificed.

The benchmark actually runs in two phases. During the first, *dry run*, phase the amount of time to accomplish a predetermined amount of work in the absence of communication is recorded. The second phase records the time for the same amount of work while the two processes are exchanging messages. The CPU availability is reported as:

$$\text{availability} = \frac{\text{time(work without messaging)}}{\text{time(work plus MPI calls while messaging)}}$$

The polling method reports message passing bandwidth and CPU availability, both as functions of the polling interval.

3.2 Post-Work-Wait Method

While the polling method yields a great deal of useful information, it does not identify implementations that violate the "progress rule" of MPI. This rule requires communication progress even if the application is involved in computation and never makes a call to the MPI library. Because the polling method makes regular, polling calls to the MPI library, it cannot uncover problems related to the progress rule. Here it is worthing noting that MPICH/GM does very well when evaluated using the polling method. The problems that we observed in the double buffering experiment are only apparent when you do not mix MPI library calls during communication.

The *Post-Work-Wait Method* mixes MPI communication and computation in a serial manner: post nonblocking MPI messages, perform computation (the work phase), and wait for the messages to complete. This strict order introduces a significant and reasonable restriction at the application level: the underlying communication system can overlap MPI communication and computation only if, after the initial MPI calls, the message passing system requires no further intervention by the application in order to progress communication. We define the term *application offload* to describe this capability. The PWW method detects whether systems exhibit application offload and identifies where host cycles are spent on communication.

Figure 4 presents a pictorial representation of the method. With respect to communication, the PWW method performs message handling in a repeated pair of operations: 1) posting non-blocking send and receive calls and 2) wait for the messaging to complete. Both processes simultaneously send and receive a single message. The worker process performs work after the nonblocking calls before waiting for message completion. The work interval is varied to effect changes in CPU availability and bandwidth.

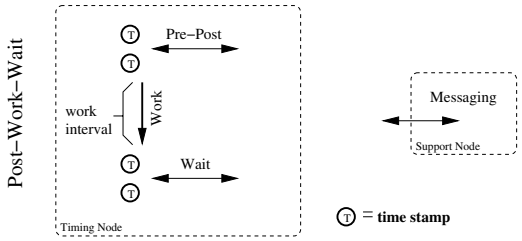

Figure 4. Post-Work-Wait Method

The PWW method collects wall clock durations for the different phases of the method. Specifically, the method collects individual durations for i) the nonblocking call phase, ii) the work phase, and iii) the wait phase. Of course, the method also records the time necessary to do the work in the absence of messaging. These phase durations are useful in identifying communication bottle necks or other causes of poor communication.

4 Sandia/Intel ASCI/Red Machine

The Sandia/Intel ASCI/Red machine[8] is the Department of Energy's Accelerated Strategic Computing Initiative (ASCI) Option Red machine. It was installed at Sandia National Laboratories in 1997 and was the first computing system to demonstrate a sustained teraFLOPS level of performance. The following briefly describes the hardware and system software environment of the machine.

4.1 Hardware

ASCI/Red is composed of more than nine thousand 333 MHz Pentium II Xeon processors connected by a network capable of delivering 400 MB/s unidirectional communication bandwidth. Each compute node contains two processors and 256 MB of main memory. Each compute node also has a network interface chip (NIC) that resides on the memory bus, allowing for low-latency access to the network.

4.2 Software

The compute nodes of ASCI/Red run a variant of a lightweight kernel, called Puma [9], that was designed and developed by Sandia and the University of New Mexico. A key component of the design of Puma is a high-performance data movement layer called Portals.

Portals in Puma are data structures in an application's address space that determine how the kernel should respond to message-passing events. Portals allow the kernel to deliver messages directly to the application without any intervention by the application process. In particular, the application process need not be the currently scheduled process or perform any message selection operations, such as tag matching, to process incoming messages. We refer to this feature as *application offload*, since the application need not be involved in the transfer of data once the operation has been set up.

In Puma, all of the resources on a compute node are managed by the *system processor*. This is the only processor that performs any significant processing in supervisor mode. The remaining processor runs application code and only rarely enters supervisor mode. This processor is called the *user processor*. This arrangement produces a slight asymmetry in the performance of the processors, but it greatly simplifies the structure of the Puma kernel and maximizes the processor cycles available to the applications.

4.3 Processor Modes

Puma supports four different modes that allow different distributions of application processes on the processors. The processor mode is determined at run-time for the processes in a parallel job when the job is launched. The following describes each of these processor modes.

The simplest processor usage mode is to run both the kernel and application process on the system processor. This mode is commonly referred to as "heater mode" since the second processor is not used and only generates heat. In this mode, the kernel runs only when responding to network events or in response to a system call from the application process. This mode does not offer any significant performance advantages to the application process.

In the second mode, message co-processor mode, the kernel runs on the system processor and the application process runs on the user processor. When the processors are configured in this mode, the kernel runs continuously waiting to process events from external devices or service system call requests from the application process. Because the time to transition from user mode, to supervisor mode, and back to user mode can be significant, this mode offers the advantage of reduced network latency and faster system call response time. Because of the increased message passing performance, this mode favors applications that are latency bound.

In the third mode, compute co-processor mode, the system processor and user processor both run the kernel and an application process. However, the kernel code running on the application processor does not perform any resource management activities, it simplify notifies the system processor when a system call is performed. The advantage of this mode is that it provides more processor cycles for the application. However, the two processors are not symmetric since the part of the application running on the shared system processor will not progress as rapidly as the portion of the application running on the dedicated user processor. In order to use this mode, the application must be use a non-standard library interface that executes a co-routine on the application processor. Because of the opportunity to utilize both processors, this mode favors applications that are compute bound.

Finally, in the fourth mode, known as virtual node mode, the system processor runs both the kernel and an application process, while the second processor also runs the kernel and a full separate application process. This mode essentially allows a compute node to be viewed by the runtime system as two independent compute nodes. The asymmetry of compute co-processor mode also exists in this mode, so the application process running on the user processor is likely to receive slightly more processor cycles than the application process running with the kernel on the system processor. This mode allows applications to avail of the user processor more easily, since the application does not need to be modified to use the non-standard co-routine interface.

In the remainder of this paper, we restrict our discussion to a comparison between standard mode (proc mode

0) and message co-processor mode (proc mode 1).

4.4 MPI Implementation

The MPI library for Puma Portals on ASCI/Red [3] is a port of the MPICH [4] implementation version 1.0.12. This implementation of MPI was validated as a product by Intel for ASCI/Red in 1997 after significant testing and has been in production use with few changes since.

The performance of the MPI implementation on ASCI/Red was studied [1] using traditional ping-pong latency and bandwidth tests. In comparison to the performance of the underlying Portals layer, MPI was shown to nominally increase latency and was able to achieve nearly identical bandwidth performance in both standard processor mode and message co-processor mode. Figure 5 shows the MPI half round trip latency performance for the standard mode (proc0) and message co-processor mode (proc1). Figure 6 shows the MPI bandwidth numbers for these processor modes.

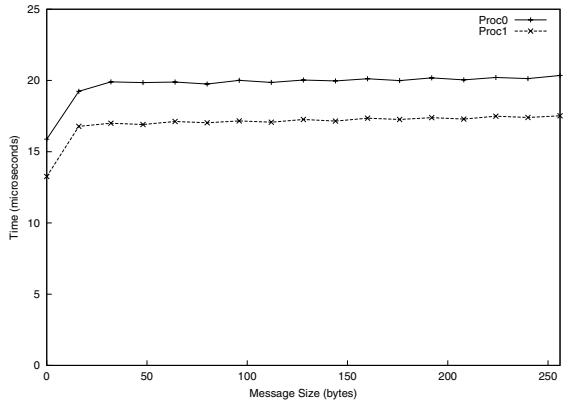

Figure 5. MPI Half Round-Trip Latency

5 COMB Results and Analysis

Because the results obtained using the ping-pong benchmark in 1997 did not uncover any unexpected performance issues, we turned our efforts to other projects. One of those projects involved the development of the COMB suite. The intent of the COMB suite was to evaluate the ability of MPI implementations to overlap computation with communication on high-end clusters, in particular systems built with programmable network interface cards like Myrinet or the Alteon Acenic Gigabit

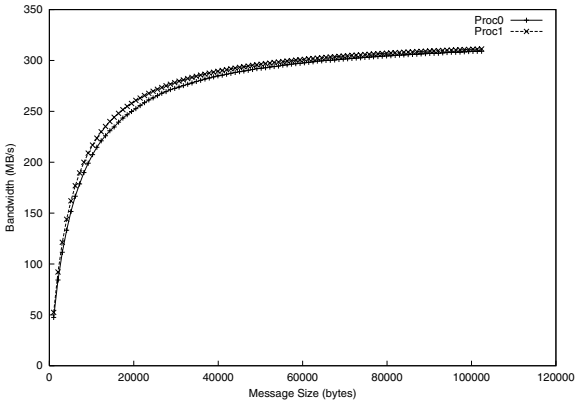

Figure 6. MPI Ping-Pong Bandwidth

Ethernet cards. On a whim, we thought it would be interesting to run the benchmarks contained in the COMB suite on ASCI/Red.

In this section we describe the performance problem that the PWW method revealed, how the MPI implementation was modified, and show the impact of this change on the performance of all of the benchmarks, including latency and bandwidth.

5.1 Initial Results

Figure 7 presents the bandwidth as a function of the work interval for 100 KB messages. This figure includes separate graphs for both processor modes. Given the differences in how bandwidth is measured in the PWW and ping-pong benchmarks, the results presented in Figure 7 are consistent with the earlier results presented in Figure 6.

In addition to bandwidth, the PWW benchmark reports the processor availability during communication. In this case, availability is reported as the ratio between the time to complete the work interval with no communication and the time to complete the work interval (and wait for message completion) while communication is progressing. Figure 8 shows CPU availability as a function of the work interval for 100 KB messages on all three processor modes.

The general shape of the curves shown in Figure 8 reflects the PWW definition of availability. When the work interval is relatively small, the work interval is too short to cover the time needed to transmit the message. This *wait while delayed* functionality suppresses apparent CPU availability until the work interval becomes sufficiently long to fill the delay period of time.

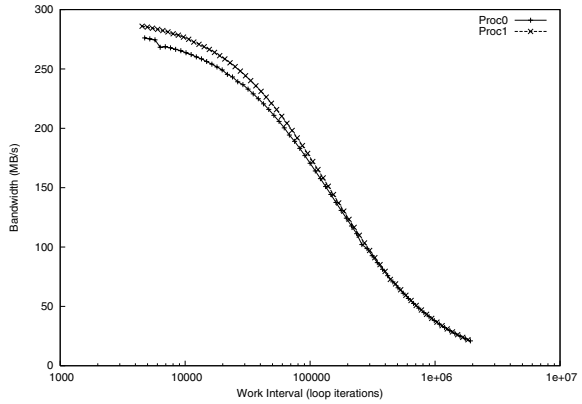

Figure 7. MPI Bandwidth for 100KB Messages

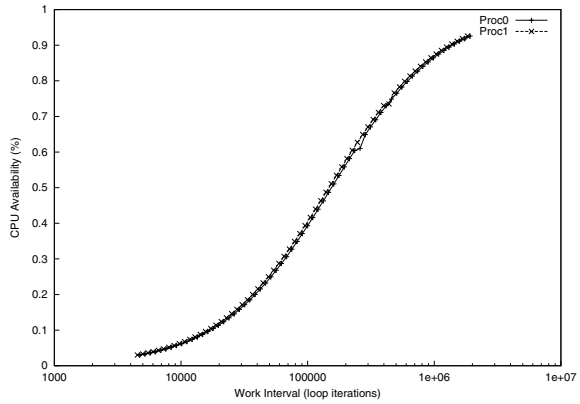

Figure 8. CPU Availability for 100KB Messages

While the shape of the curves was expected, we were surprised by the fact that there was no separation between these curves. Given that message passing is entirely handled by the system processor in message co-processor mode, we had expected that the availability would be significantly higher

Further analysis of the data provided by the PWW benchmark identified the source of the problem. In particular, the PWW benchmark provides the time taken to post each message. In message co-processor mode, we would expect that the time to post a message would have little or no dependence on the size of the message, since the application process can make a request to the kernel to start the transfer and then return to computation. However, when we looked that the posting times for various message sizes, we saw that PWW was reporting a direct relationship between the post time and the length of the message. Figure 9 shows the post time for four message sizes across varying work intervals in message co-processor mode.

The results in Figure 9 indicated that the MPI non-blocking operations were waiting for the data transfer to be completed before returning from the library. A quick inspection of the MPI implementation confirmed that non-blocking MPI send calls would trap to the kernel for the data transfer request and then immediately wait for the kernel to complete the transfer before returning from the call. By doing this, the MPI library eliminates any possibility of overlapping computation with sending messages.

This limitation does not have any effect in standard mode, since the kernel must complete the data transfer before returning control the application process anyway. The standard ping-pong benchmark used to measure latency and bandwidth does not reveal this problem either, since it only evaluates network performance and does not consider processor overhead. For these reasons, the loss of opportunity to overlap in message co-processor mode went undetected.

5.2 MPI Enhancement

The MPI implementation was modified to return immediately after making a send request to the kernel. This change involved moving the structure that indicates send completion from the local stack into the send request structure. Rather than waiting for completion of the send immediately after making the request, the MPI implementation checks or waits for completion in the *MPI_Test() or MPI_Wait()* family of functions. In message co-processor mode, this gives the kernel an opportunity to transfer the data while the application process continues computing.

5.3 Impact of the Change

We now present several results that show the impact of this small change on latency, bandwidth, overhead, and CPU availability for the different processor modes in Puma. In some cases, this enhancement led to significant gains in performance.

Figure 10 presents the processor availability graph for using the modified MPI implementation. We now

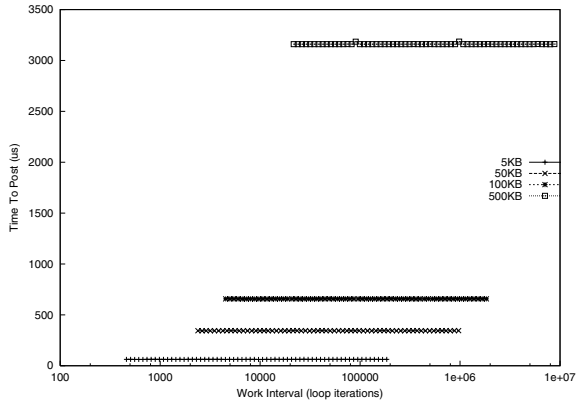

Figure 9. Time to Post in Message Co-Processor Mode

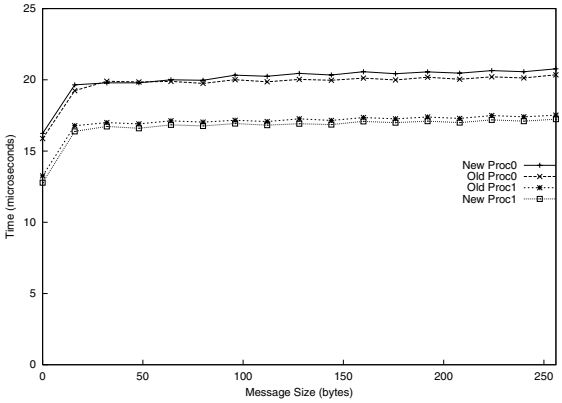

Figure 11. New MPI Half Round-Trip Latency

see the improvement in processor availability for message co-processor mode that we had expected to see.

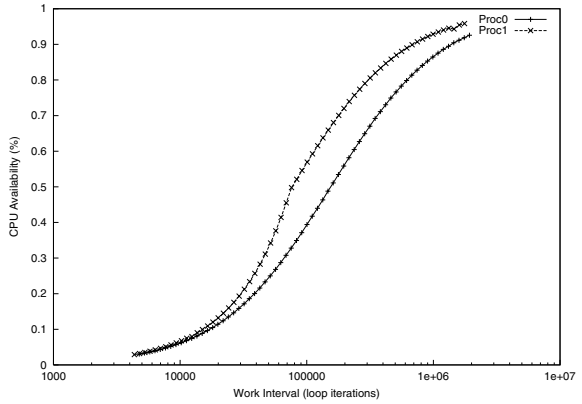

Figure 10. CPU Availability for 100KB Messages

Figure 11 compares the MPI half round trip latency performance of the previous implementation with the new implementation for both processor modes. The results are mixed. For message co-processor mode, latency was improved by about 1 μsec. However, in standard mode, the new implementation is about 1 μsecworse.

Figure 12 compares the MPI bandwidth performance of the previous implementation with the new implementation. The numbers are nearly identical for both modes. For standard mode, the performance of the new implementation exceeds the old one at around 3 KB. In message co-processor mode, the numbers are virtually identical.

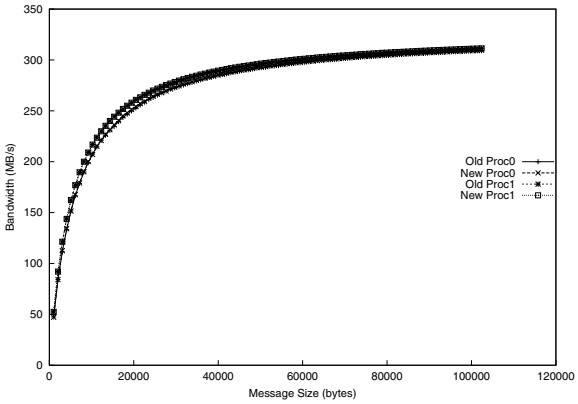

Figure 12. New MPI Ping-Pong Bandwidth

6 Discussion

While the COMB benchmark suite was initially developed to compare MPI implementations for cluster systems, we thought it might be interesting to run the benchmarks on ASCI/Red. The PWW benchmark was a valuable tool that revealed a significant performance problem with the MPI implementation on ASCI/Red. We believe this benchmark to be a valuable tool in measuring message passing performance relative to proces-

sor availability. We have demonstrated its ability to expose and help diagnose performance problems that other traditional message passing benchmarks do not.

References

[1] R. Brightwell and D. S. Greenberg. Experiences implementating the MPI standard on sandia's lightweight kernels. Technical Report SAND97-2519, Sandia National Laboratories, August 1997.

[2] R. Brightwell, T. Hudson, R. Riesen, and A. B. Maccabe. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[3] R. Brightwell and L. Shuler. Design and implementation of MPI on Puma Portals. In *Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[5] W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell. Comb: A portable benchmark suite for assessing MPI overlap. Technical Report TR-CS-2002-13, Computer Science Department, The University of New Mexico, April 2002.

[6] A. B. Maccabe, W. Lawry, C. Wilson, and R. Riesen. Distributing application and OS functionality to improve application performance. Technical Report TR-CS-2002-11, Computer Science Department, The University of New Mexico, April 2002.

[7] Myricom, Inc. The GM message passing system. Technical report, Myricom, Inc., 1997.

[8] Sandia National Laboratories. *ASCI Red*, 1996. http://www.sandia.gov/ASCI/TFLOP.

[9] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[10] J. B. White and S. W. Bova. Where's the overlap?: An analysis of popular mpi implementations. In *Proceedings of the Third MPI Developers' and Users' Conference*, March 1999.